ICC 2017
WASHINGTON
JULY 02-07, 2017

Extracting the Essential Cartographic Functionality of Programs on the Web

Florian Ledermann

Research Group Cartography, Department for Geodesy and Geoinformation, TU Wien, Vienna, Austria
florian.ledermann@tuwien.ac.at

Abstract: Following Aristotle, F. P. Brooks (1987) emphasizes the distinction between "essential difficulties" and "accidental difficulties" as a key challenge in software engineering. From the point of view of cartography, it would be desirable to identify the *cartographic essence* of a program, and subject it to additional scrutiny, while its *accidental* properties, again from the point of view of cartography, are usually of lesser relevance to cartographic analysis. In this paper, two methods that facilitate extracting the cartographic essence of programs are presented: close reading of their source code, and the automated analysis of their runtime behavior. The advantages and shortcomings of both methods are discussed, followed by an outlook to future developments and potential applications.

Keywords: Cartographic programming, web cartography, methods, cartographic transformations

1. Introduction

Innovation in digital cartography and GIS is historically tightly connected to the creation of software through programming (Monmonier, 1985; Chrisman, 2006). Recently, in web cartography, JavaScript has become the prime language in which to express cartographic programs, and the shift to open standards and the desire to employ and develop the full range of cartographic representation and interaction techniques has lead to the rise of programs and tools that implement the whole cartographic "stack" – from loading and processing geodata to generalization, symbolization and rendering – in the web browser, potentially with the assistance of ready-made "building blocks" provided by third-party APIs such as Google Maps or D3 (Ledermann and Gartner, 2015; Roth et al., 2014). Creating such cartographic programs, but also judging the quality and relevance of interactive cartographic programs shared online, and relating what is implemented there to the knowledge of cartographic techniques, becomes an increasingly relevant challenge for cartographers.

Following Aristotle, F. P. Brooks (1987) identifies distinguishing "essential difficulties" and "accidental difficulties" as a key challenge in software engineering, with continuous progress made towards removing non-essential difficulties through abstraction, so that only essential aspects remain for the programmer to be concerned with. From the point of view of cartography, it would be desirable to learn about the *cartographic essence* of a program, and subject it to additional scrutiny, while its *accidental* difficulties, again from the point of view of cartography (such as the specificities of dealing with the underlying technical platform), are usually of lesser relevance to cartographic analysis. Arguably, what is considered essential from a cartographic point of view may differ greatly from what is considered most relevant from a generic software engineering perspective. Therefore, the paradigms developed by software engineers to deal with the complexity of a program may be ill suited for cartographers to allow them to engage with the software and find what is essential from their point of view well-represented in the code. At the same time, an overwhelming amount of the code may be dealing with essential engineering aspects (such as network access, memory management or error handling) that are of lesser interest – and potentially confusing – to the cartographer.

Attempts to extract or construct a formalized essence of cartography can be found in research on cartographic ontologies (Iosifescu-Enescu and Hurni, 2007; Smith, 2010; Penaz et al., 2014) as well as works proposing taxonomies for specific aspects of cartography, such as generalization (McMaster and Shea, 1992; Foerster et al., 2007) or interaction (Roth, 2013), and theoretical disciplines such as spatial information theory or analytical cartography (Moellering, 2000). In the context of web cartography and "neocartography", which is shaped by a diverse range of actors not solely from academic but also commercial and community-driven backgrounds (Kraak, 2011), we face the situation that programs and tools created by these parties do not necessarily follow the concepts established by academic cartography. Inspecting the source code of a cartographic program, we may fail to clearly identify concepts from the cartographic literature or curriculum. Concepts may be named differently in the code, out of unawareness of the cartographic vocabulary or because of a different perspective on the processes from an engineering standpoint; Names of variables and functions may be deliberately obfuscated to prevent such analysis in commercial products; Procedures may not be as trivial in the pragmatic context of a running system as in its pure abstract formulation in theory (See Bostock and Davies (2013) for a discussion of the practical complexities involved in an implementation of the simple but versatile projection of a straight line segment onto the map).

To learn about the cartographic essence contained in a cartographic program, an empirical, bottom-up approach seems desirable – how can we identify and "reverse engineer" (Chikofsky and Cross, 1990) the essence of cartographic program code written by others? With such a method, we would hope to derive findings from an analysis of actual program codes that are more *fine-grained* than general taxonomies of cartographic techniques, more *universally* representing the full cartographic process than formalized operators known from GIS and other sub-disciplines, and more *distinct* and semantically meaningful than the raw source code itself.

In the remainder of this paper, two novel methods for analyzing the essence of program code with respect to cartographic transformations will be presented: close reading by humans, and automated runtime analysis. For both methods, a basic framework of analysis, grounded in Tobler's idea of *transformations* as the fundamental paradigm in cartography (Tobler, 1979; Chrisman, 1999), is used. The basic questions initially guiding the investigation are: where can we find transformations in the code, and where do we locate these transformations in the overall cartographic pipeline from raw data to physical output?

2. Close reading of cartographic program code

Analyzing the source code of a program as a textual artifact by "close reading" has been attempted by social sciences scholars (Berry, 2011; Mackenzie, 2006), but not to our knowledge in the domain of cartography or with a specific interest in cartographic transformations. Our method for analyzing cartographic programs through close reading is loosely based on the method of *content analysis*, a social sciences method for extracting concepts from human communication artifacts (Krippendorff, 2012). Content analysis builds on an incremental process of "*coding*"[1], i.e. highlighting and assigning keywords to parts of the text, line by line, in close reading sessions.

```
svg.selectAll(".place-label")
    .data(places.features)
  .enter().append("text")
    .attr("class", "place-label")
    .attr("transform", function(d) { return "translate(" + projection(d.geometry.coordinates)
    .attr("x", function(d) { return d.geometry.coordinates[0] > -1 ? 6 : -6; })
    .attr("dy", ".35em")
    .style("text-anchor", function(d) { return d.geometry.coordinates[0] > -1 ? "start" : "en
    .text(function(d) { return d.properties.name; });
```

Fig. 1. Part of the code of a cartographic program written using D3, annotated by close reading. This snippet contains aspects of visual element creation (purple), iteration (orange), cartographic projection (yellow), geodata access (brown), conditions (red) and assigning visual variables (blue).

[1] The "coding" activity of content analysis is not to be confused with programming which is sometimes causally referred to by the same verb.

This method has been applied to a corpus of example programs using three different base technologies – the Google Maps API, D3 (Bostock et al., 2011) (both based on JavaScript) and the Kartograph API (a hybrid API with JavaScript and Python functionality). A first set of programs has been chosen from the collection of "official" examples found on the APIs' web pages; while these programs may differ greatly in functionality between the individual technologies, we assume that their creators were competent in programming using the respective technology, and that these examples "showcase" the specific strengths of the technology. A second set of programs implements the creation of a simple choropleth map in each of the base technologies. Where available, implementations by the API's creators were preferred, in order to assure the program author's competence with the given technology.

Applying our method to a collection of cartographic programs results in i) a taxonomy of transformations and operations identified in these programs and ii) an annotated corpus of program code, both of which can be further analyzed by quantitative or qualitative means. For example, for programs implementing a simple choropleth map, code for the Google Maps API is concerned to 51% with *accidental* difficulties and 69% with *essential* cartographic functionality[2], while a similar result is achieved using D3 having to deal with only 27% of *accidental* difficulties – supporting to some extent the claim that D3 is an "elegant" API that allows users to focus on the essential parts of a visualization. For further details, interested readers are referred to the initial results published in (Ledermann, 2016).

The main drawback of the close reading method is that it requires substantial effort by well-trained humans to analyze program code. Analyzing a single example program by close reading took the author between one and two working days on average – given the already limited number of professionals well trained in cartography *and* programming, this is hardly a sustainable approach to analyze large corpora of cartographic programs. Furthermore, subjective bias cannot be ruled out in tasks that require human judgment such as the assignment of cartographic semantics to abstract source code. To mitigate this limitation, each artifact would be required to be examined by multiple readers, verifying intersubjective congruency by measures such as "intercoder agreement" or α-agreement (Krippendorff, 2012). Findings could further be verified by evaluating a large number of programs, which is, however, difficult on a practical level due to the amount of work involved.

An alternative strategy would be to look for automated methods to analyze the cartographic transformations present in a cartographic program, either as a sufficient analysis in itself or as a way to identify the cartographically "interesting" parts of the code that would be candidates for subsequent analysis by close reading of a much smaller piece of code. Such an approach will be investigated in the next section.

3. Automatic analysis of cartographic programs

Program code follows a strict syntax that can be unambiguously dissected into its components (*tokenization*) and transferred into a data structure representing all statements and expressions contained in the code (*parsing*). So besides viewing the program as text, one can generate a more abstract representation of the program from that text, close to how a computer would actually represent the program in memory before running it, that contains all its information in a machine-processable form. Thus, we can transform the program code into a data structure that should be more accessible to automated analysis of its behavior.

However, reasoning about code without executing it is provably limited – fundamentally, the Church/Turing hypothesis (Church, 1936; Turing, 1937) states that for a program it is not generally possible to determine whether it will terminate, without actually running the program. While some limited techniques for "static analysis" have been developed in software engineering (Nielson et al., 2004), the inability to reliably determine which parts of a program will be executed at runtime remains a major obstacle to using static analysis techniques for any analysis that relates to the result of a program (in our case: the generated map) – it can simply not be inferred from the code whether a given part of that code will ever actually run. Furthermore, JavaScript as a language has some properties that make it particularly ill-suited to static analysis (Andreasen and Møller, 2014).

In contrast to static analysis, *dynamic analysis* is performed by running the program and simultaneously monitoring its behavior. For dynamic analysis, it is necessary to inject some kind of instrumentation into the runtime environment the program is run in (in our case, the web browser), in order to perform the monitoring of the program's

[2] The sum is larger than 100% because in some lines both accidental and essential functionality are present.

operations. This instrumentation attempts to capture all operations of the program that are relevant for the given analysis task.

The subject of this investigation being cartographic transformations, we are interested in a program's *visual* output and all transformations that have been applied to that output. To capture this information, all visually relevant operations of the program are intercepted and recorded. Web browsers offer a plethora of methods to create or modify the visual appearance of a page – we currently identified 128 methods and properties provided by the Document Object Model (DOM), the web browsers basic API for interacting with the page, that need to be intercepted. To capture transformations, the program is executed step by step, tracing every variable and each operation performed on them.

Using this technical fixture, all visual elements generated by the program can be captured, decomposed into their visual attributes (e.g. a line has x and y coordinates for start- and endpoint, a thickness value and a color value as a minimum set of attributes), and for each attribute the sequence of operations that have been involved in its calculation (its *data flow graph*) can be retrieved. For each operation in the data flow graph, in turn, its location in the source code can be retrieved and visualized on demand for subsequent human inspection (potentially by selectively applying the *close reading* technique presented in section 2 on selected parts of the code). See figure 2 for an illustration of the result of such an analysis for a single visual attribute of a single point on the map.

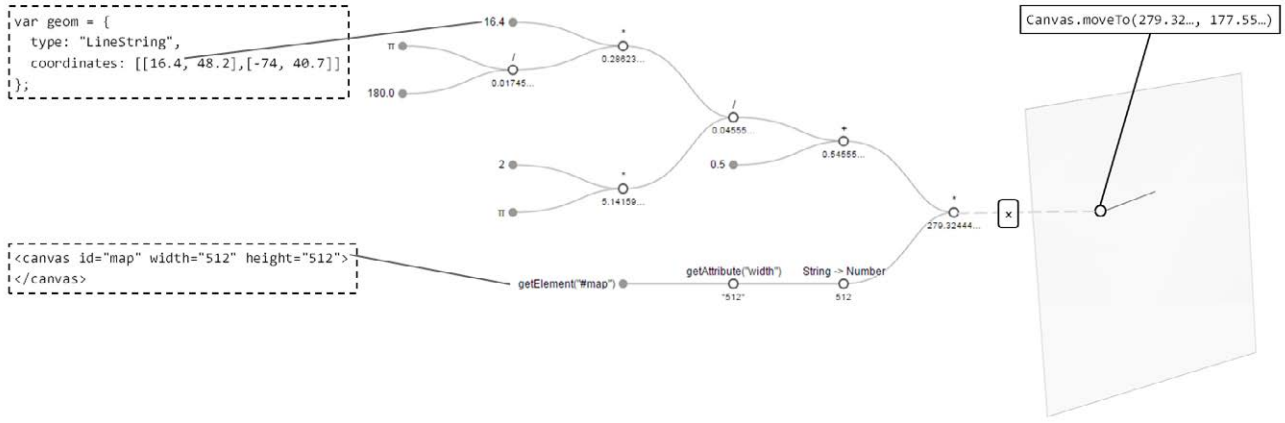

Fig. 2. The visualization of parts of the cartographic *essence* of a program as a result of the automatic analysis. Depicted here is a data flow graph of a single graphical attribute (the x coordinate of the endpoint of a line, derived from the longitude of a given point by an implementation of the Web Mercator projection). Each node in the data flow graph is either an arithmetic expressions or data conversion, and can be traced back to its location in the source code, facilitating closer inspection on demand.

Dynamic program analysis provides a method to reduce potentially vast programs, containing a lot of "accidental" functionality of no primary interest to cartographers, to a list of operations which are potentially relevant for cartographic analysis (i.e. visual operations). For each of these operations, a data flow graph is provided that details how the concrete value (for example, a coordinate or color value) was calculated from operations defined in the code. Both of these results (the list of operations, and the data flow graph for each operation/attribute) have promising potential for subsequent automatic or manual analysis that should be much easier to perform than the analysis of the raw source code, since it will operate on a highly aggregated representation of a small selection of the original code.

An immediate next step would be to attempt to identify patterns in the data flow graphs. For example, identifying well known projections in the graph should be possible, and would work independent of the concrete syntax and semantics of the implementation of the projection.

4. Discussion and Outlook

Extracting the cartographic essence of programs is a challenging task with many possible approaches. Attempts that use the textual representation of a program's source code as basis for analysis (such as close reading or static analysis) are heavily reliant on the structure and semantics of the code, which may be guided by software engineering requirements or the technological context and not primarily by cartographic concerns. Such methods are therefore sensitive to syntactic variations in the code that may not change the underlying transformations, but provide a structurally different implementation (For example, between different APIs or even with an updated version of an API). They work well in identifying cartographic patterns in code using well-designed APIs, but break down with obscure implementations or unknown paradigms. Because of their reliance on program semantics (e.g. the names of functions and variables), these approaches are also vulnerable to code obfuscation, a technique frequently employed by commercial software producers to impede reverse engineering of their products[3].

The proposed method of dynamic analysis reliably captures all visually relevant operations of a program run, and traces each attribute of these operations back through the transformations defined in the code. This approach should therefore be much more robust to syntactic and semantic variations, i.e. different implementations of the same cartographic operation (projection, interpolation, generalization etc.), and should produce identical or similar data flow graphs for the same cartographic concept implemented in different APIs, API versions or even deliberately obfuscated programs. However, these assumptions about the proposed method are still to be verified in detail and across a larger variety of "real life" programs.

The next step towards an automatic analysis system would be the identification of patterns in the dataflow graph that represent well known cartographic operations. An initial task would be to determine whether a given program is a cartographic program at all – whether it contains any cartographic transformation. This of course depends on the concrete definition of a cartographic transformation, but one could require, for example, a projection from geographic space to map space to be involved. Identifying cartographic programs would therefore involve looking for patterns representing cartographic projections in their dataflow graphs.

Although the practical and theoretical limitations of the manual close reading approach were discussed and seem to prohibit a sweeping application on large corpora, further validation of the method by verifying intercoder agreement across individuals, both for experienced programmers as well as for novices or students, seems desirable. Even with automatic analysis methods in place, the selective study of source code will be required for some of the final steps in an analysis, and the role of source code comprehension in learning (cartographic) programming needs to be better understood (Roth et al., 2014).

If automatic and semi-automatic methods such as the ones proposed in this paper can be developed into a working system, further practical applications can be envisioned. On the level of individual maps and users, a web browser plugin for analyzing online maps with respect to their cartographic processes could be developed, assisting novices and professionals in scrutinizing such programs. For teaching purposes, an interactive learning environment could be envisioned that visualizes and explains the cartographic processes of a program, providing a visual and domain-specific interface to the program code. Finally, an improved understanding of the cartographic essence of programs could lead to a refinement in our conceptualizations of cartographic transformations, which in turn could lead to the development of more powerful and/or usable APIs for interactive cartography.

Acknowledgements

The author wants to thank Georg Gartner and Silvia Klettner for fruitful discussions and helpful suggestions in early stages of this research.

[3] Please be advised that applying reverse engineering methods on commercial software may be a breach of the law and/or the usage terms of the software and consult with a legal advisor before attempting to use similar techniques on such programs!

References

Andreasen, E., Møller, A., 2014. Determinacy in Static Analysis for jQuery, in: Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).

Berry, D.M., 2011. The Philosophy of Software: Code and Mediation in the Digital Age. Palgrave Macmillan, Basingstoke.

Bostock, M., Davies, J., 2013. Code as Cartography. The Cartographic Journal 50, 129–135. doi:10.1179/0008704113Z.00000000078

Bostock, M., Ogievetsky, V., Heer, J., 2011. D3: Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics 17, 2301–2309. doi:10.1109/TVCG.2011.185

Brooks, F.P., Jr., 1987. No Silver Bullet – Essence and Accidents of Software Engineering. Computer 20, 10–19. doi:10.1109/MC.1987.1663532

Chikofsky, E.J., Cross, J.H., 1990. Reverse engineering and design recovery: a taxonomy. IEEE Software 7, 13–17. doi:10.1109/52.43044

Chrisman, N., 1999. A transformational approach to GIS operations. International Journal of Geographical Information Science 13, 617–637.

Chrisman, N.R., 2006. Charting the Unknown: How Computer Mapping at Harvard Became GIS. ESRI Press.

Church, A., 1936. A note on the Entscheidungsproblem. Journal of Symbolic Logic 1, 40–41. doi:10.2307/2269326

Foerster, T., Stoter, J., Kobben, B., 2007. Towards a formal classification of generalization operators, in: Proceedings of the 23rd International Cartographic Conference (ICC2007). Presented at the ICC2007, Moscow, Russia.

Iosifescu-Enescu, I., Hurni, L., 2007. Towards cartographic ontologies or "how computers learn cartography," in: Proceedings of the 23rd International Cartographic Conference (ICC2007). Presented at the ICC2007, Moscow, Russia.

Kraak, M.-J., 2011. Is there a need for neo-cartography? Cartography and Geographic Information Science 38, 73–78.

Krippendorff, K., 2012. Content Analysis: An Introduction to Its Methodology. SAGE.

Ledermann, F., 2016. Initial Findings from Close Reading of Cartographic Programs, in: Workshop „Code Loves Maps", AGILE 2016. Helsinki, Finland.

Ledermann, F., Gartner, G., 2015. mapmap.js: A Data-Driven Web Mapping API for Thematic Cartography, in: Proceedings of the 27th International Cartographic Conference (ICC2015). Presented at the ICC2015, Rio de Janeiro, Brasil.

Mackenzie, A., 2006. Cutting Code: Software and Sociality. Peter Lang Publishing, New York.

McMaster, R.B., Shea, K.S., 1992. Generalization in digital cartography. Association of American Geographers, Washington, DC.

Moellering, H., 2000. The Scope and Conceptual Content of Analytical Cartography. Cartography and Geographic Information Science 27, 205–224. doi:10.1559/152304000783547858

Monmonier, M.S., 1985. Technological Transition in Cartography. University of Wisconsin Press, Madison, Wisconsin.

Nielson, F., Nielson, H.R., Hankin, C., 2004. Principles of Program Analysis. Springer Science & Business Media.

Penaz, T., Dostal, R., Yilmaz, I., Marschalko, M., 2014. Design and Construction of Knowledge Ontology for Thematic Cartography Domain. Episodes 37, 48–58.

Roth, R.E., 2013. An empirically-derived taxonomy of interaction primitives for interactive cartography and geovisualization. IEEE Transactions on Visualization and Computer Graphics 19, 2356–2365. doi:10.1109/TVCG.2013.130

Roth, R.E., Donohue, R.G., Sack, C.M., Wallace, T.R., Buckingham, T.M.A., 2014. A Process for Keeping Pace with Evolving Web Mapping Technologies. Cartographic Perspectives 25–52. doi:10.14714/CP78.1273

Smith, R.A., 2010. Designing a cartographic ontology for use with expert systems, in: Proceedings of the 18th AutoCarto Conference. Presented at the 18th AutoCarto Conference, Orlando, USA.

Tobler, W.R., 1979. A Transformational View of Cartography. The American Cartographer 6, 101–106.

Turing, A.M., 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society S2.42, 230–265. doi:10.1112/plms/s2-42.1.230